US 20050050387A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2005/0050387 A1**

Mariani et al. (43) **Pub. Date: Mar. 3, 2005**

(54) **DEPENDABLE MICROCONTROLLER, METHOD FOR DESIGNING A DEPENDABLE MICROCONTROLLER AND COMPUTER PROGRAM PRODUCT THEREFOR**

(75) Inventors: **Riccardo Mariani**, Pisa (IT); **Silvano Motto**, Viareggio (IT); **Monia Chiavacci**, Prataccio (IT)

Correspondence Address:
SEED INTELLECTUAL PROPERTY LAW GROUP PLLC
701 FIFTH AVE
SUITE 6300
SEATTLE, WA 98104-7092 (US)

(73) Assignee: **Yogitech Spa**, Viareggio (IT)

(21) Appl. No.: **10/888,355**

(22) Filed: **Jul. 9, 2004**

(30) **Foreign Application Priority Data**

Jul. 11, 2003 (EP) 03015860.4

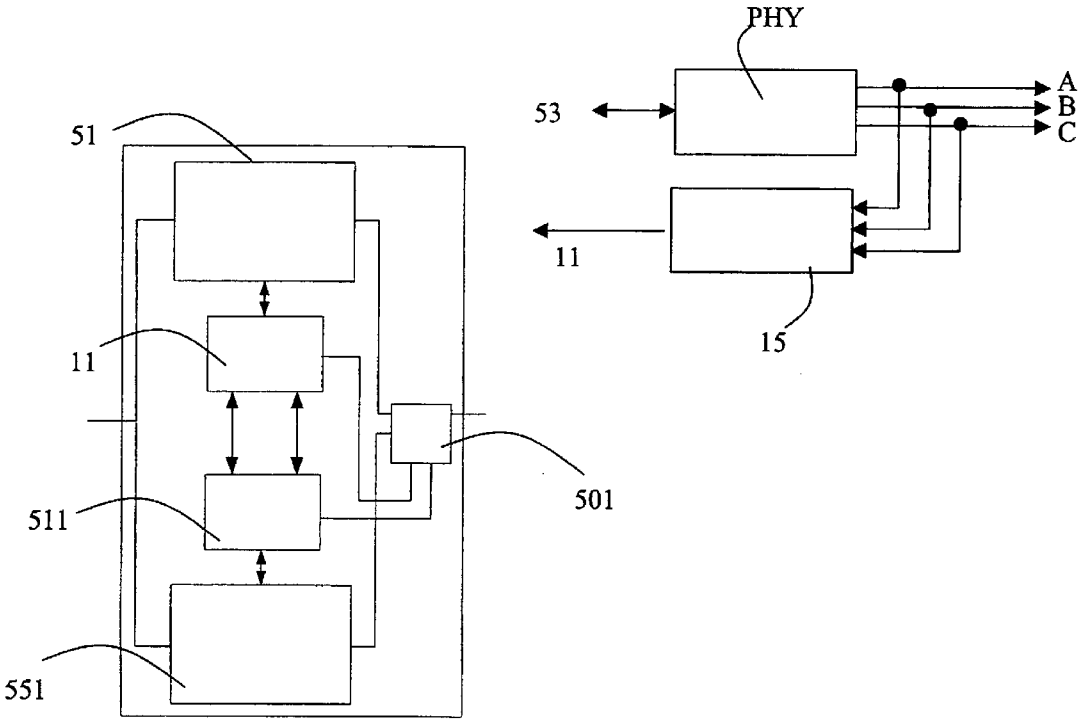
Publication Classification

(51) **Int. Cl.**[7] **G06F 11/00**

(52) **U.S. Cl.** **714/13**

(57) **ABSTRACT**

A microcontroller comprising a central processing unit and a further fault processing unit suitable for performing validation of operations of said central processing unit. The further fault processing unit is external and different with respect to said central processing unit and said further fault processing unit comprises at least a module for performing validation of operations of said central processing unit and one or more modules suitable for performing validation of operations of other functional parts of said microcontroller. Validation of operations of said central processing unit is performed by using one or more of the following fault tolerance techniques: data shadowing; code&flow signature; data processing legality check; addressing legality check; ALU concurrent integrity checking; concurrent mode/interrupt check. The proposed microcontroller is particularly suitable for application in System On Chip (SoC) and was developed by paying specific attention to the possible use in automotive System On Chip. The invention also includes a method for designing and verify such fault-robust system on chip, and a fault-injection technique based on e-language.



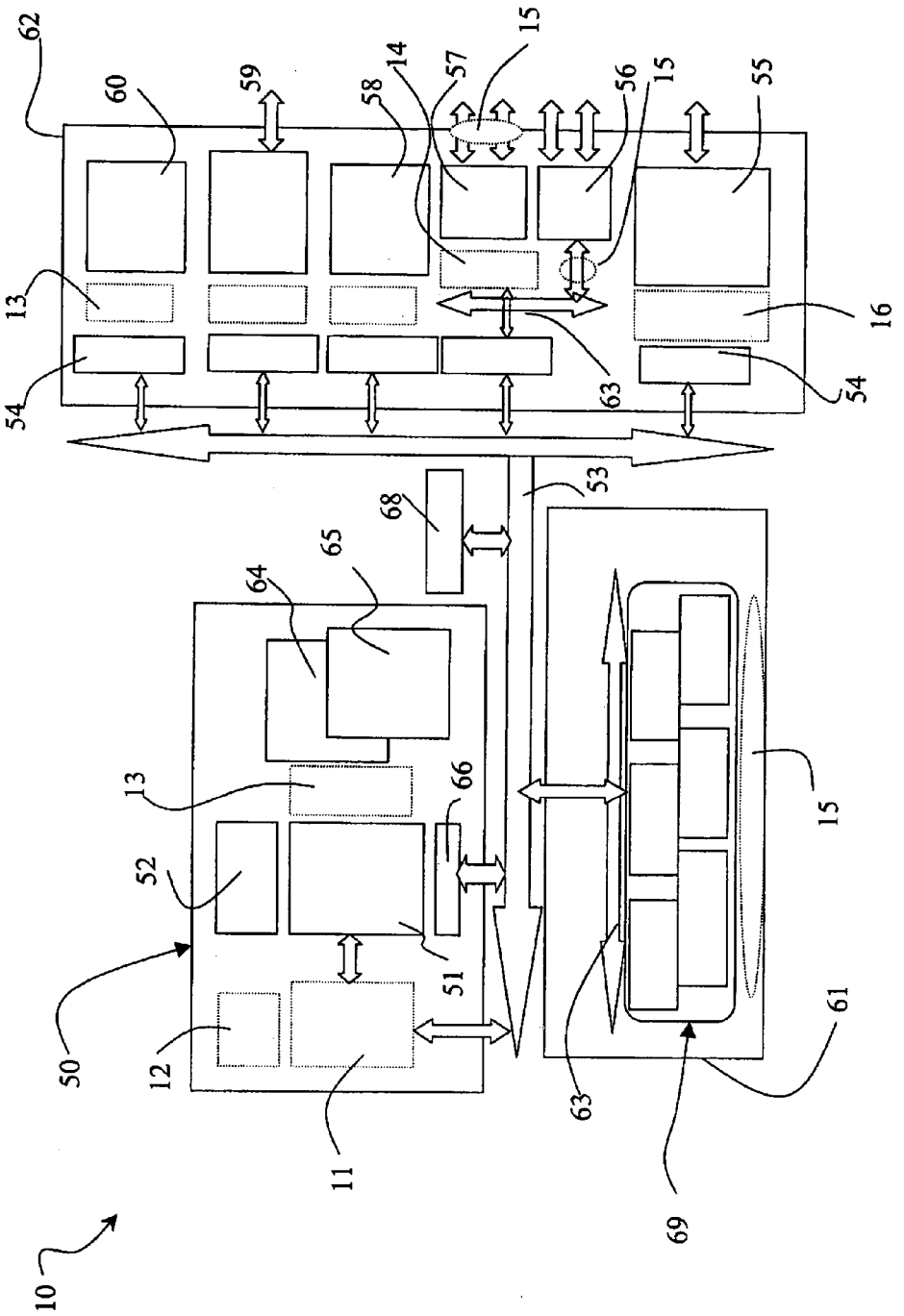

62
60
59
58
14
57
15
56
15
55
13
16
54
63
54
53
68
65
64
13
66
52
15
51
61
50
12
63
11
69
10

FIG. 1

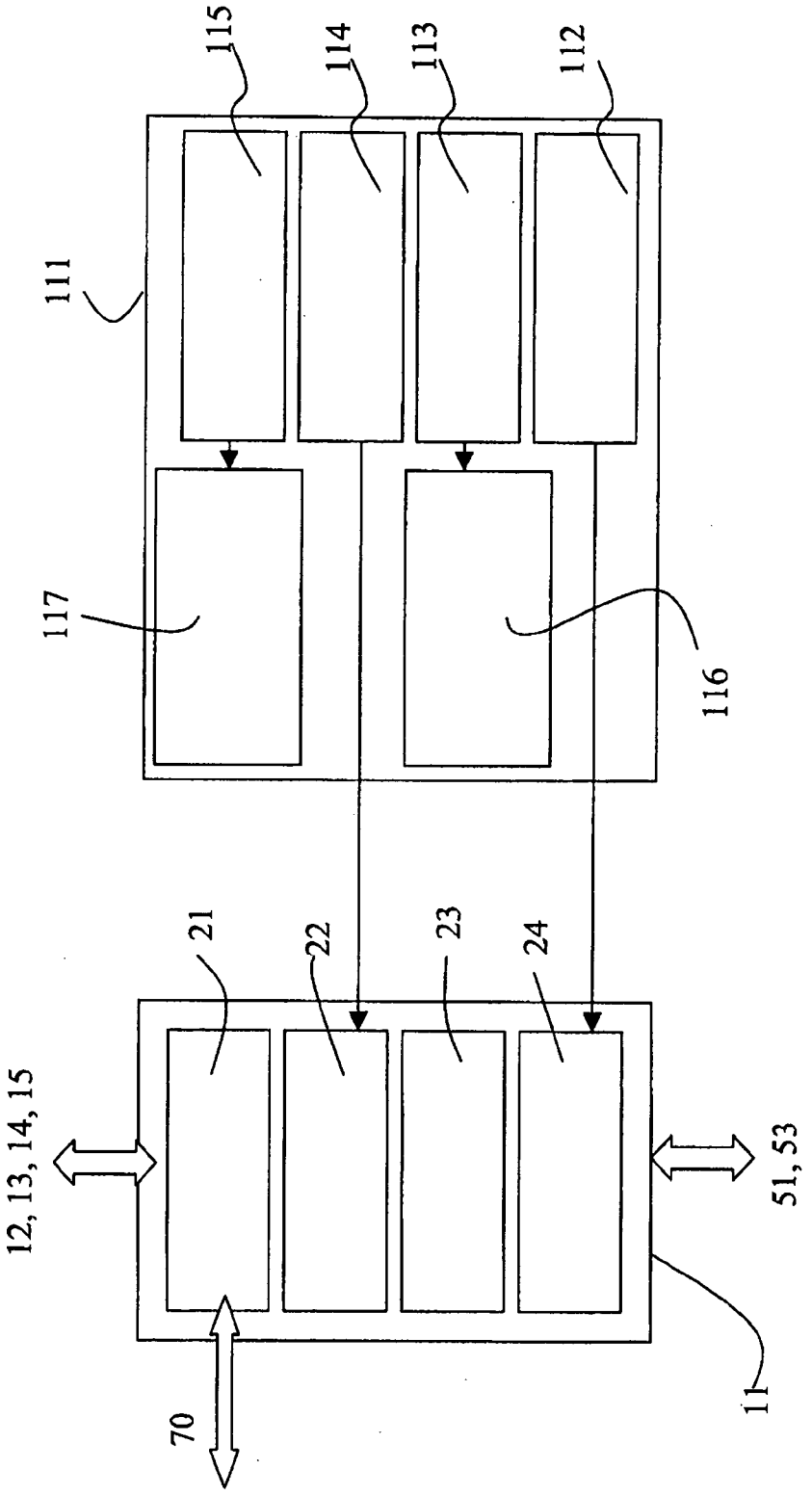


FIG. 2

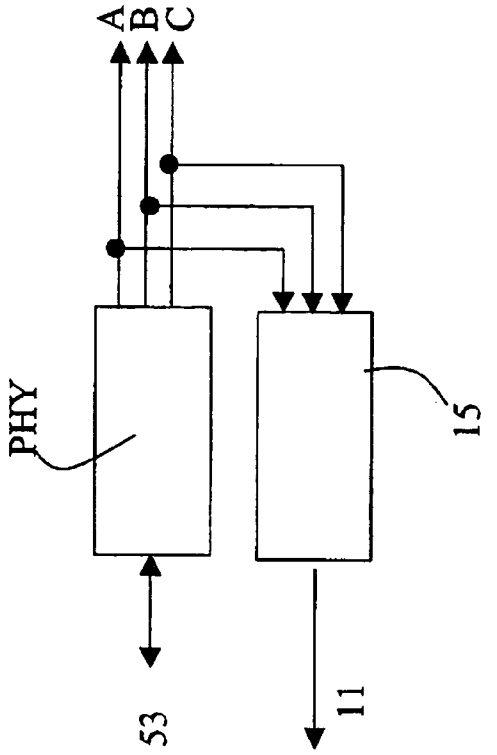


FIG. 4

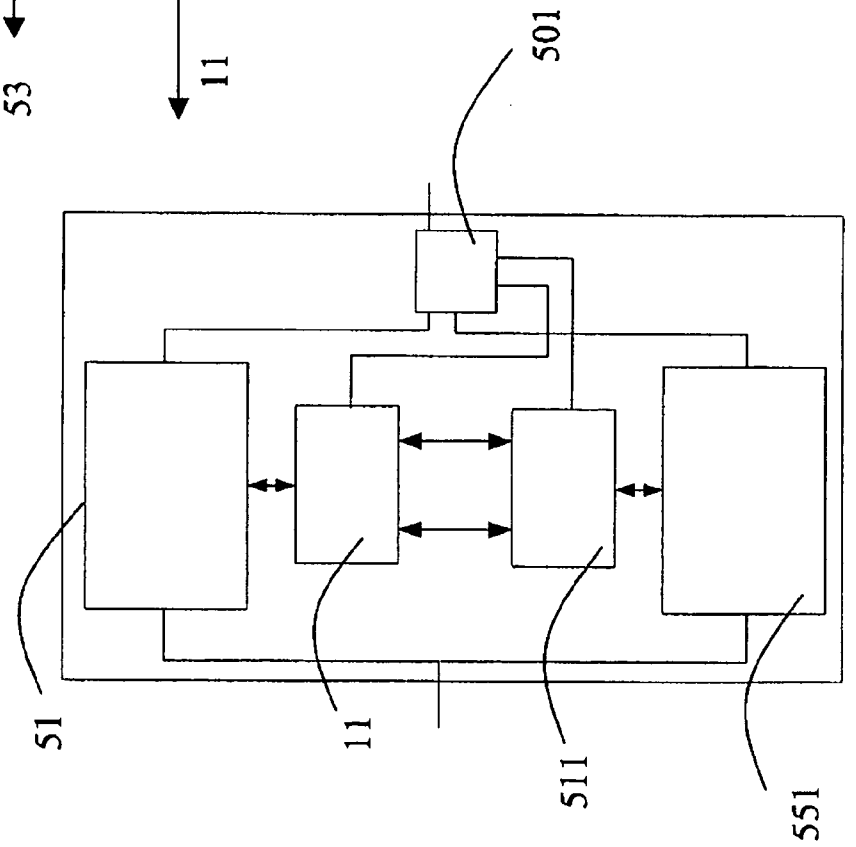


FIG. 3

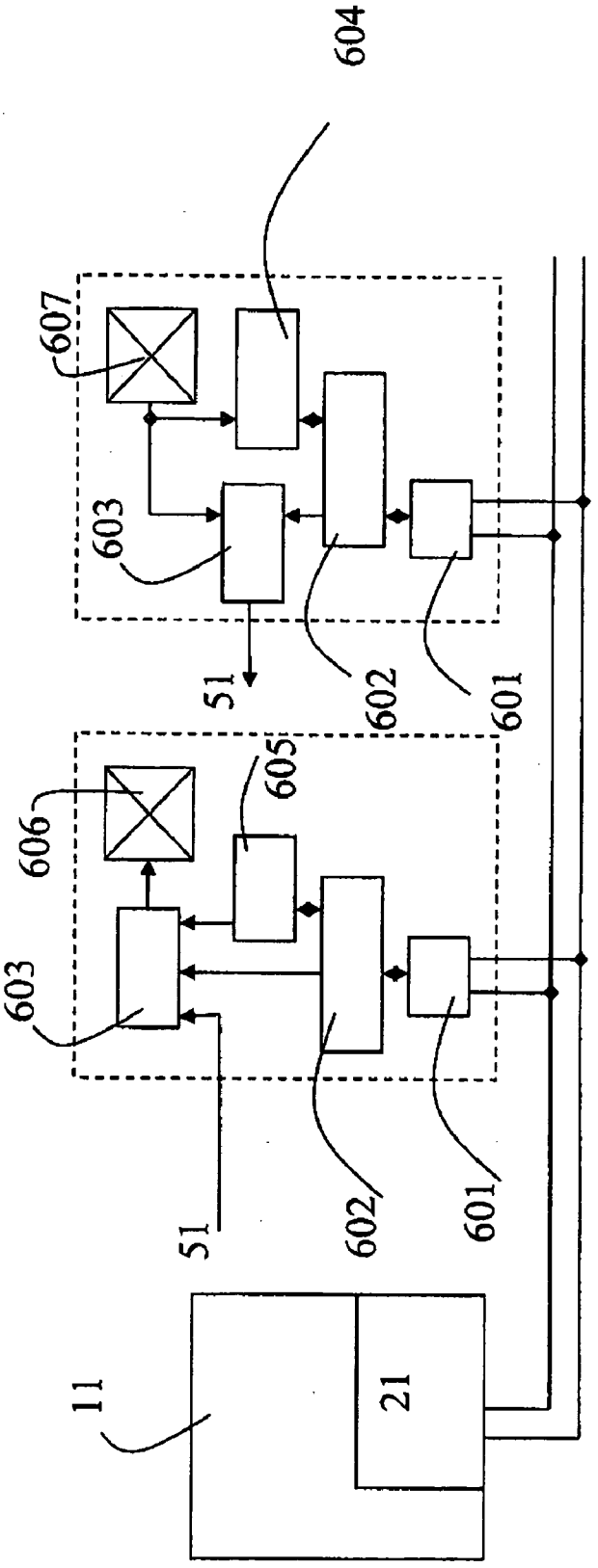


FIG. 5

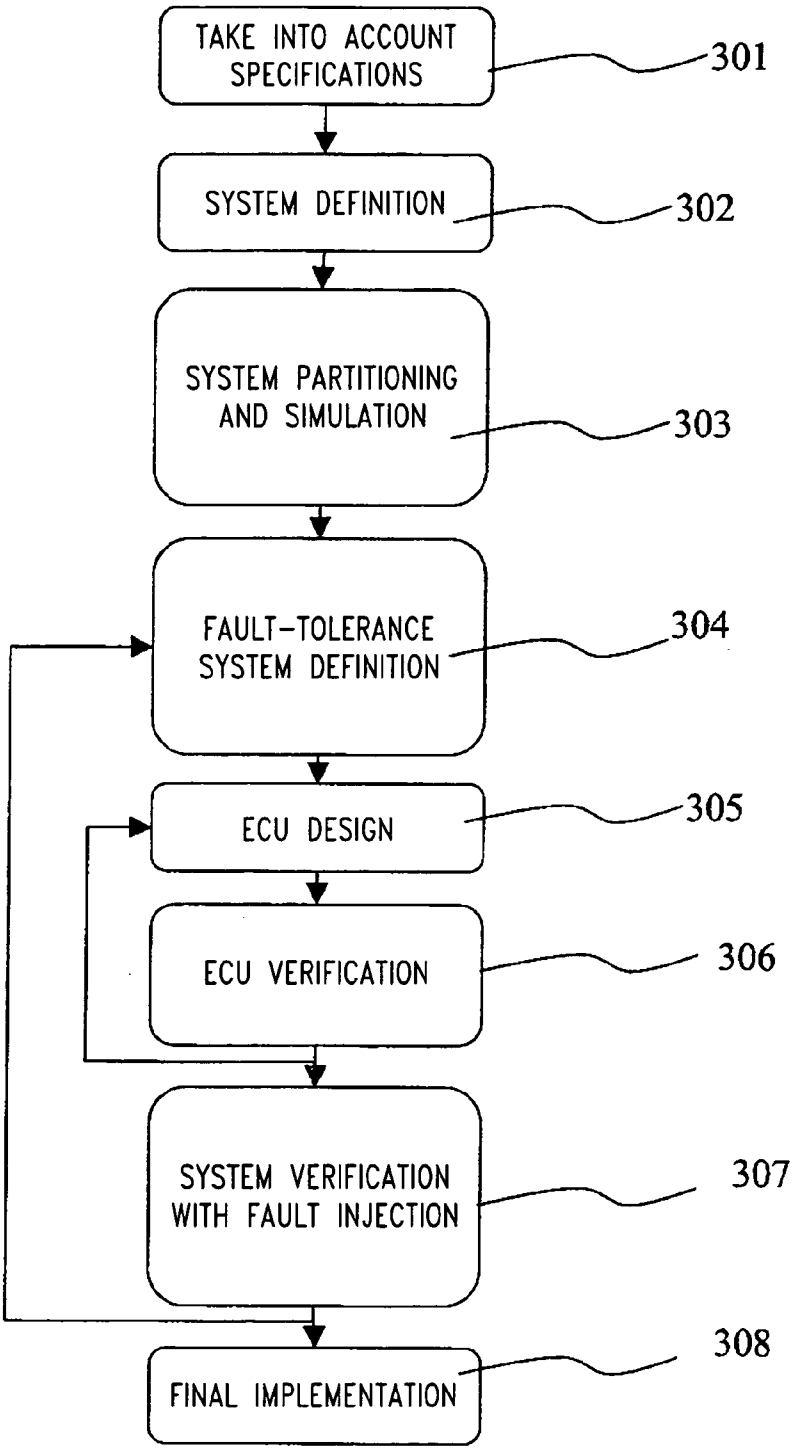

TAKE INTO ACCOUNT SPECIFICATIONS
301
SYSTEM DEFINITION
302
SYSTEM PARTITIONING AND SIMULATION
303
FAULT-TOLERANCE SYSTEM DEFINITION
304
ECU DESIGN
305
ECU VERIFICATION
306
SYSTEM VERIFICATION WITH FAULT INJECTION
307
FINAL IMPLEMENTATION
308

FIG. 6

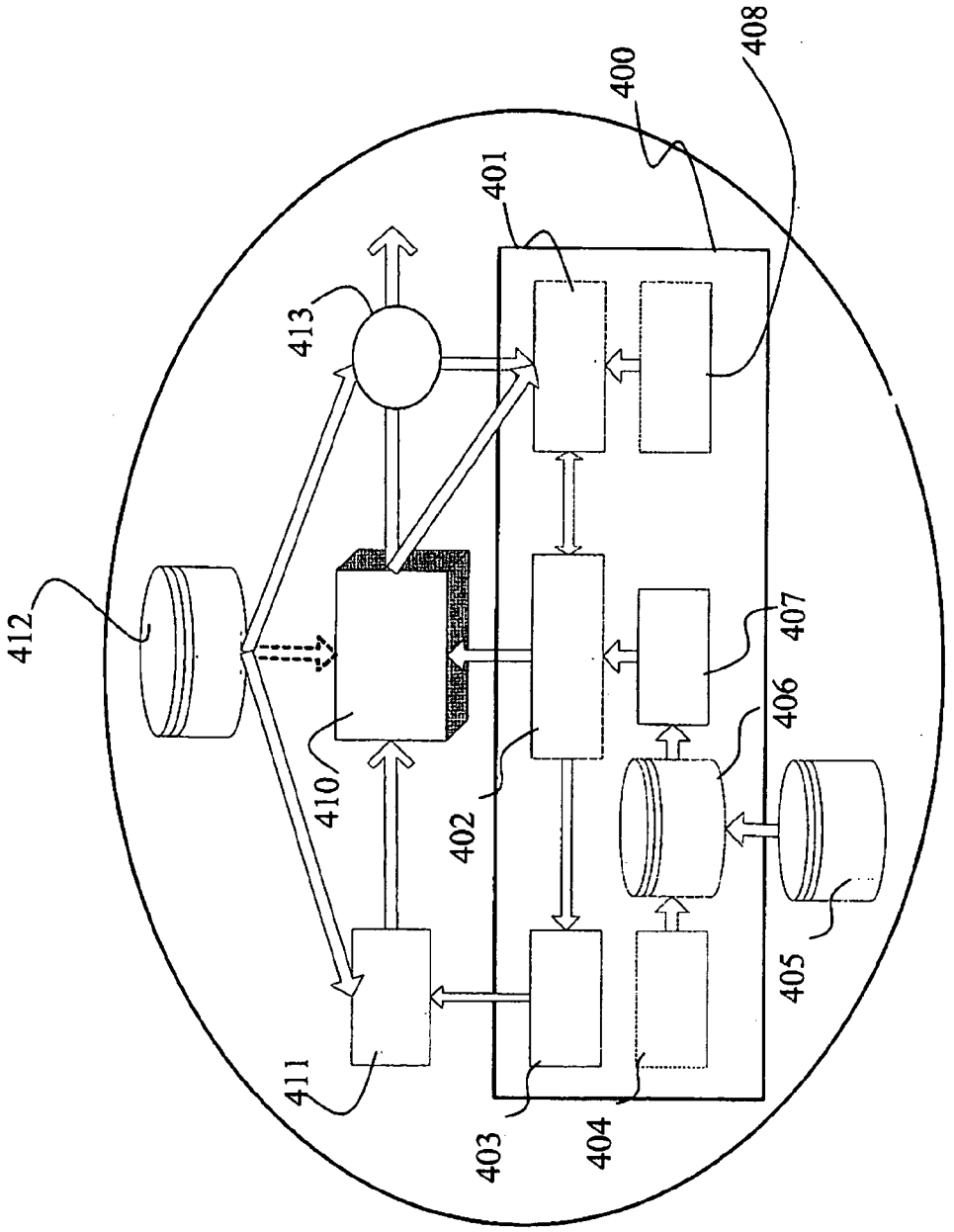


FIG. 7

DEPENDABLE MICROCONTROLLER, METHOD FOR DESIGNING A DEPENDABLE MICROCONTROLLER AND COMPUTER PROGRAM PRODUCT THEREFOR

BACKGROUND OF THE INVENTION

[0001] 1. Field of the Invention

[0002] The present invention relates to techniques for achieving dependability in microcontrollers units (microcontroller) and was developed by paying specific attention to the possible use in automotive systems.

[0003] Exemplary of use of such microcontrollers are systems for anti blocking systems (ABS), torque control system (TCS) and automatic stability control (TSC). However, this is not to be construed in a limiting sense of the scope of the invention that is in fact applicable to a wide variety of applications, comprising x-by-wire, infotainment, biomedical applications, communications, and so on.

[0004] The term 'microcontroller' in the following description should be intended in a broad sense, i.e., a System On a Chip (SOC) comprising analog to digital and digital to analog converters, CPU and CPU peripherals, memories, system buses, internal or external sensors and/or transducers, controls, instrumentations, output devices.

[0005] The present invention also relates to techniques and corresponding computer program products for the design of dependable microcontrollers.

[0006] 2. Description of the Related Art

[0007] In recent times, the implementation on vehicles of microelectronic systems in order to increase efficiency, safety, performance and comfort as well as information and entertainment has considerably grown.

[0008] Such microelectronic systems are based on central processing units and interconnected using robust networks, provided with means for the detection of faults. Such robust networks remove the need for thousands of costly and unreliable wires and connectors, used to make up a wiring loom.

[0009] Of course such systems must be highly reliable, as failure will cause fatal accidents. However, the car is a hostile environment having a wide range of temperatures and subject to dirt, salt spray, dust, corrosive fluids, heavy vibration and sudden shock, electrical noise, electromagnetic interference. Further it is needed that such microelectronic systems show near zero power consumption when the car is parked in order not to consume battery charge.

[0010] These requirements must combine with requirements for near perfect system modeling and verification such that system behavior is predictable and fail-safe under the most extreme conditions.

[0011] As a consequence the use of complex System-On-Chip in the automotive field has been limited, as compared to consumer and communications applications.

[0012] It is thus apparent the importance of having a design platform capable to help the system designer to design more complex ECU (Embedded Computational Unit) in less time and with the highest reliability. In general, the most important requirements for a design platform are the abilities to:

[0013] allow hardware and software standardization;

[0014] be adaptable at the customer's needs;

[0015] generate clean codes & scripts in a well defined flow;

[0016] be easily linkable with the operating system;

[0017] allow easy verification of sub-blocks and custom blocks;

[0018] be upgradeable.

[0019] In particular, for automotive applications, it is also important:

[0020] scalability in a fast and easy way;

[0021] to give verification detail and allow customizability of the verification;

[0022] to be fault-robust and allow fault-injection in early stage (both for design and verification);

[0023] In order to allow a better understanding of the description that follows some definitions pertaining fault-robustness are here supplied.

[0024] In general it is called a 'failure' the event that occurs when the delivered service of a system deviates from the specified service. It is called 'error' the part of the system state which is liable to lead to failure. The phenomenological cause of the error is called the 'fault'. 'System dependability' is defined as the quality of the delivered service such that reliance can justifiably be placed on this service.

[0025] Dependability is evaluated on the basis of the measure of the three following quantities:

[0026] Reliability: the probability, quantified in MTTF (Mean Time To Failure) that a piece of equipment or component will perform its intended function satisfactorily for a prescribed time and under specified environmental conditions;

[0027] Availability: the probability, quantified by MTTR/MTTF (Mean Time To Repair) that the system will be functioning correctly at any given time;

[0028] Safety: the freedom from undesired and unplanned event that results (at least) in a specific level or loss (i.e., accidents).

[0029] To achieve a dependable system, the following methods can be used, separately or together:

[0030] fault-avoidance, that provides for avoiding faults by design;

[0031] fault-removal, that provides for reducing the presence of faults, by verification. In this method fault-injection techniques are fundamental;

[0032] fault-tolerance, a technique that provides correct operation despite presence of faults;

[0033] fault-forecasting or fault-evasion, that provides for estimating, by evaluation, the presence, the creation and the consequences of faults and taking pre-emptive steps to stop the fault from occurring.

[0034] From the previous definitions, it is worth noting that fault-tolerance by itself does not solve the dependability needs of a complex system. For instance, a fault-tolerant

system could not be fail-safe. Thus, in the following, reference will be made to the concept of robustness, defined as the ability of a system to continue to function despite the existence of faults in its component subsystems or parts, even if system performance may be diminished or otherwise altered—but always in a safe way—until the faults are corrected. A fault-robust system will keep its function even with changes in internal structure or external-environment.

[0035] The solutions known from the prior art provide for hardening the fabrication technology, so that is less prone to radiation-induced soft errors, or for introducing redundancy: at gate level, for example by implementing triple flip-flop structures with majority voting (N-modular redundancy); or at CPU level, for example by introducing some coding in the logic units of the processor; or at microcontroller level, for example by using multiple processors running in step with watchdogs; or at software level, for example by using multiple software threads, or a mixture of all of the above techniques.

[0036] Technology hardening and redundancy at gate level are very expensive in terms of area and performance overhead, CPU redesign time and so on.

[0037] Redundancy at CPU level is less expensive in terms of area overhead, but it requires CPU redesigning.

[0038] Redundancy at microcontroller level is the most used technique, either using N-modular redundancy (N=2 is the most used, i.e., dual redundancy) or with dynamic redundancy. From the U.S. patent application No. 2002/00777882 a dual-redundant microcontroller unit is known, comprising a first central processing unit and a secondary processing unit coupled to the first processing unit. A functional comparison module is coupled to the primary processing unit and to the secondary processing unit for comparing a primary output of the primary processing unit and a secondary output of the secondary processing unit to detect a fault if the primary output does not match the secondary output. Functional comparison is performed by analyzing signatures of the outputs. Signatures are computed in idle cycles or are obtained as results of test sequences or are determined on the basis of external address and data buses of the CPU itself, so that only a limited visibility of the content of the CPU is available. This very often results in inefficient signatures from the point of view of fault coverage and memory occupation, and in a greater latency of error detection.

[0039] Dual redundant techniques, as depicted also in German patent application DE 19933086A1, of course imply a great increase in circuits size (at least the double) and an even greater increase in production costs. Moreover, performances are affected because of need for slower clocks, increased gate count and inclusion of redundant software.

[0040] Dynamic redundancy techniques are also known that require only the processor plus a fault detector that can identify faulty behavior of the processor. Such techniques allows for a higher utilization of computing resources, but they could generate a greater latency, because of the greater number of computations required to achieve good fault coverage. The most common solutions in this case consist in watchdogs or very simple fault detectors monitoring the data and address bus only to compute simple consistency checks. In other known solutions, the CPU itself is charged to handle part of the dependability issues, interacting with a simple external controller: in U.S. Pat. No. 5,436,837, a microcomputer and a monitoring module are disclosed. The monitoring module is preferably configured as a gate-array which executes a sequence control of the microcomputer. The monitoring module determines the correct or defective operation of the microcomputer from a comparison of the results of this processing.

[0041] Also solutions are known, for example from U.S. Pat. No. 5,880,568, that introduce redundancy at software level only. Such solutions however affect strongly the microprocessor performance because of the fault-detection tasks.

BRIEF SUMMARY OF THE INVENTION

[0042] One of the goals of the present invention is thus to provide a microcontroller that achieves high dependability, while making proper use of computing resources and circuit size.

[0043] According to the present invention, this object is achieved by means of a dependable microcontroller having the characteristics set forth in the claims that follow. The invention also relates to a corresponding method for designing a dependable microcontroller, as well as to a computer program product directly loadable in the memory of a digital computer and comprising software code portions for performing the methods of the invention when the product is run on a computer.

[0044] Substantially, the solution according to the invention provides a microcontroller that contains a fault tolerant processing unit for validating operation of the central processing unit. The fault tolerant processing unit implements also a distributed fault tolerant detection method on the chip through suitable interfaces and fault tolerant hardware blocks. A method for designing the system comprising the fault tolerant processing unit, interfaces and hardware blocks is also provided.

[0045] In comparison with prior art arrangements, the proposed solution reduces the area overhead required on the chip, allows low latency control of the fault in the central processing unit and it is fully and flexibly customizable according to the System On Chip and user needs.

BRIEF DESCRIPTION OF THE DRAWINGS

[0046] The present invention will now be described, purely by way of non-limiting example, with reference to the annexed drawings, wherein:

[0047] **FIG. 1** shows a block diagram of the microcontroller of the invention;

[0048] **FIG. 2** shows a block diagram of a detail of the microcontroller of **FIG. 1**;

[0049] **FIG. 3** shows a block diagram representing a further embodiment of the microcontroller of the invention;

[0050] **FIG. 4** shows a further detail of the microcontroller of **FIG. 1**, related to an hardware verification component;

[0051] **FIG. 5** shows a further detail of the hardware verification component shown in **FIG. 4**;

[0052] **FIG. 6** shows a block diagram pertaining to a design procedure of the microcontroller of **FIG. 1**;

[0053] **FIG. 7** shows a block diagram representing a fault injection procedure used in the design procedure of **FIG. 6**.

DETAILED DESCRIPTION OF THE INVENTION

[0054] The basic idea underlying the microcontroller described herein is that, in order to have a fault-robust systems, hardware redundancy is implemented, controlled and configured by software in order to achieve fault tolerance of complex CPU-based system on a chip without compromising execution time and code dimensions and with an acceptable hardware overhead.

[0055] The microcontroller according to the invention implements distributed fault-tolerant techniques in every part of the system to be monitored, through suitable circuitry that is demanded to watching a particular problem and communicating with a central fault-tolerant processor, that is external and different from the central processing unit, i.e., it is not a dual central processing unit and it has autonomous computing resources external to that of said central processing unit. Such a central fault-tolerant processor coordinates all the peripherals and keeps track of the fault-history of the system.

[0056] In **FIG. 1 a** block diagram is shown representing the hardware architecture of a microcontroller **10**.

[0057] Such a microcontroller **10** basically comprises a processor **50**, that can be for instance a RISC unit like an ARM processor, comprising a central processing unit core **51**, associated to a coprocessor **52**. The processor **50** comprises also a data memory module **64**, instruction cache memories **65** and a AMBA (Advanced Microcontroller Bus Architecture bus interface) interface **66**. A CPU system bus **53** is provided for connecting the processor **50** with an interfaces module **61**, comprising among others a general purpose input/output interface, a serial parallel interface, a real time clock, a UART, timers and watchdogs. These interfaces are indicated collectively with reference **69**.

[0058] Also a vector interrupt controller **68** is shown, connected to the CPU system bus **53**.

[0059] Such a CPU bus system **53** also allows communication with a set of peripherals **62**. Such a set of peripherals **62**, comprises application specific peripherals **55**, analog interfaces **56**, bus interfaces **57**, a direct memory access (DMA) module **58**, an external bus interface **59**, an embedded memory **60**. The interfaces module **61** and the set of peripherals **62** also include suitable peripheral buses **63**, interconnecting their different devices.

[0060] All the components mentioned above are typical of a microcontroller architecture, and they will be not described in further detail, as their functions and use are well known to a man skilled in the art of microcontroller designing.

[0061] According to the invention, the microcontroller **10** further comprises a Fault-Recognition Accelerator **11**, that is a processor operating as the main fault-manager. In the following, for conciseness' sake, the Fault-Recognition Accelerator **11** will be sometimes also referenced by its acronym, FRAC. Such a Fault-Recognition Accelerator **11** is composed by a plurality of user-configurable modules for implementing different fault-detection and fault-tolerance techniques. Fault-recognition Accelerator **11** and related modules and peripherals are indicated in dashed line in **FIG. 1** for easier identification.

[0062] The Fault-Recognition Accelerator **11** is placed, in the processor **50**, in communication with the central processing unit core **51**. The Fault-Recognition Accelerator **11** operates in association with a Fault Recognition Memory **12**, also placed in the processor **50**, that can be a dedicated memory or a part of the CPU memory with dedicated ports. The Fault Recognition Memory **12** stores data, such as code, historical tracks and other data needed by the Fault-Recognition Accelerator **11** to implement its functions.

[0063] As better detailed in a following part of the description, the Fault Recognition Accelerator **11** comprises different modules, and it is dedicated to survey the overall system and the other fault-tolerant peripherals through the system bus **53** and other dedicated signals. The Fault-Recognition Accelerator **11** is also responsible for start-up and periodic system built-in tests. Fault Recognition Accelerator **11** can be accessed both through the processor core **51**, a JTAG port of the system and also directly through bus interfaces, in order to allow fast diagnostic and maintainability at system level.

[0064] In order to attain full dependability the Fault Recognition Accelerator **11** is able to generate fault alarms and events, like an exception, a program retry or a system stop, due to a fault in the central processing unit **51** or a fault in the Fault Recognition Accelerator **11** itself. The Fault Recognition Accelerator **11** must be always able to react to these problems and avoid unsafe system conditions.

[0065] Fault-Tolerant Memory interfaces **13** are also provided on the microcontroller **10**, for interfacing with memories such as data memory **64**, cache memories **65** comprised in the processor **50**, the embedded memory **60** and the DMA module **58** comprised in the set of peripherals **62**. Faul-Tolerant Memory interfaces **13** are composed by various user-configurable modules implementing different memory fault-detection and fault-tolerance techniques like Error Correction Code (ECC), Built-In Self Repair (BISR), and so on.

[0066] For the protection of bus such the peripheral bus **63** or of the CPU system bus **53** or other-external buses such CAN bus, a Fault-Tolerant bus interface controller **14** is provided.

[0067] As it can be observed in **FIG. 1**, hardware Verification Components **15** (hVC) are also provided, in association with different modules and devices of the microcontroller **10**. In the embodiment shown a hardware Verification Component **15** is thus associated to the interfaces module **61**, to the analog interfaces **56** and to bus interface **57**. The hardware Verification Components **15** comprise various user-configurable modules implementing fault-detection and fault-tolerance techniques directly derived from the system-level verification requirements of a verification platform and a corresponding design and verification procedure, that will be detailed in the following with reference to **FIGS. 4 and 5**.

[0068] The microcontroller **10** also comprises a Remote Fault Recognition Accelerator **16**, associated to the Application Specific Peripherals **55**, that is a special processor for application specific safety requirements.

[0069] The hardware architecture of Fault Recognition Accelerator **11** is represented in **FIG. 2**. The Fault Recognition Accelerator **11** is associated to a Fault-Recognition Compiler **111**, that is a software process for creating initialization/configuration boot routines for the Fault Recognition Accelerator **11** and for extracting runtime settings to be inserted in the user code in order to facilitate the interaction between the CPU core **51** and the Fault Recognition Accelerator **11** itself. Moreover, if area overhead is an issue or if the CPU core **51** is not able to interface with Fault Recognition Accelerator **11** in a proper way, Fault Recognition Compiler **111** can also insert in the user code various software modules, similar to modules of Fault Recognition Accelerator **11** for providing the same fault-detection and fault-tolerance functionalities.

[0070] The Fault Recognition Accelerator **11** associated to the Fault-Recognition Compiler exploits a mixed "data and circuit-oriented" instead than a pure "circuit-oriented" fault protection mechanism, i.e., the protection is firstly applied to part of the system and part of the data that are relevant for the application. This relevant part is identified both by the user itself in the process of designing the System on Chip, as it will be detailed with reference to **FIG. 6**, and by the Fault Recognition Compiler **111** through some techniques of code analysis as it will be detailed with reference to **FIG. 7**.

[0071] Therefore, the fault protection is obtained with a combination of techniques implemented both in hardware, in the Fault Recognition Accelerator **11**, and in software, in the Fault Recognition Compiler **111**.

[0072] In **FIG. 2** such modules for Fault Recognition Accelerator **11** and Fault Recognition Compiler **111** are shown.

[0073] The Fault Recognition Accelerator **11** thus comprises:

[0074] System Modules **21**, that are modules needed to interact with the Fault recognition peripherals such as Fault recognition Memory **12**, Bus Interface Controller **14**, Fault-Tolerant Memory interfaces **13**, hardware verification components **15**, and to interact with basic system components such as clock and power supplies, indicated collectively with reference **70**. Moreover, system modules **21** include basic modules needed to manage all the other internal modules of the Fault Recognition Accelerator **11** and the program watchdog. System modules **21** can be regarded as CPU independent for what concerns design; however they depend on the overall system architecture;

[0075] Main Protection Modules **22**, that are modules needed to implement the main fault protection techniques, such as Variable Legal/Delta values, Code&Flow Signature, and so on. These modules can be regarded as CPU independent;

[0076] CPU Generic Modules **23**, that are modules needed to process data arriving from Fault Recognition Accelerator/CPU interface modules **24** and to exchange data with the main protection modules. CPU Generic Modules **23** are slightly CPU dependent, i.e., basic algorithms are CPU independent, however the way the data are processed can change with the CPU family.

[0077] Fault Recognition Accelerator/CPU interface modules **24**, that are modules needed to exchange data with the CPU core **51** and the CPU system bus **53**. Modules **24** are CPU dependent.

[0078] In **FIG. 2** is also shown the Fault Recognition Accelerator compiler **111**, comprising different software modules, organized as in the following:

[0079] Initialization Modules **112**, which are modules needed to initialize all the basic functions of the Fault Recognition Accelerator **11** and initialize its peripherals, independently by an Operative System **116** or by a User Application Code **117**;

[0080] Operative System Interaction Modules **113**, that are modules needed to interact with the operative system;

[0081] Application Configuration Modules **114**, that are modules needed to configure the Fault Recognition Accelerator **11**, e.g., the runtime signatures, based on the Operative System **116** and on the User Application Code **117**;

[0082] Run Time Modules **115**, that are modules needed to insert into the Application Code **117** the needed instruction to allow fault-detection and protection techniques. As it will be better detailed in a following part of the description, most of the run time modules **115** are the counter-part of hardware modules comprised in the Fault Recognition Accelerator **11**, i.e., they are needed only if the correspondent function is not carried out in the correspondent hardware module, for instance because of area overhead constraints or interface capabilities of the Fault Recognition Accelerator/CPU interface modules **24**.

[0083] The Fault Recognition Accelerator **11**, in order to detect faults in the CPU core **51**, executes three main different kinds of tasks in different times during the microcontroller activity, in cooperation with other FRAC peripherals:

[0084] microcontroller fault-forecasting at power-on: a sequence of BIST tests, or BISTs, run by the Fault Recognition Accelerator **11** after power-on. Such BISTs are self-consistent software run in the Fault Recognition Accelerator **11** or in a piece of dedicated hardware therein and they are focused either to cover the highest percentage of the functionality under test or to cover only the core functionalities needed by the real application. A BIST sequence performed by the Fault Recognition Accelerator **11**, substantially comprises a Built In Self Test on the RAM of the Fault Recognition Accelerator **11**, that is performed after power-on through its embedded Fault-tolerant Memory Interfaces **13**. Then a Built In Self Test on the Fault Recognition Accelerator **11** is performed. Subsequently, validation passes on the CPU core **51** and a CPU RAM Built In Self Test is performed through Fault-tolerant Memory Interfaces **13**. Then in a Built In Self Test on the CPU core **51** is performed. Subsequently, fault-tolerant peripherals are validated through dedicated hardware Verification Components **15**. Then a BIST operation is performed on the bus interfaces through dedicated fault-tolerant bus interface **14**. Finally external BIST

operations are launched, to reach a 'System Ok' condition, if all above steps are successfully passed;

[0085] microcontroller on-line fault-forecasting: it is a subset of previously described BISTs executed during the system runtime. The microcontroller on-line self-test is run in two ways:

[0086] during idle times, by isolating the block to be tested (e.g., the CPU or the RAM or one of the peripherals) by the rest of the system saving the current state, then by running the correspondent BIST and finally restoring the initial state;

[0087] during normal operation, the Fault Recognition Compiler can be allowed to insert (for instance during loops), extra dummy codes in order to do in parallel active-tests in part of the CPU currently not used.

[0088] on-line fault tolerance: it is a set of techniques for detecting faults during normal operation and taking actions when a fault is found, including data shadowing, code and flow signature, data processing legality check, addressing legality check, ALU concurrent integrity checking and concurrent mode/interrupt check.

[0089] In the following the techniques for implementing on-line fault tolerance will be detailed.

[0090] A data shadowing procedure is implemented, that is relevant for the operation of the proposed microcontroller, as it is applied both to variables wherever they are located in the microcontroller, and to CPU registers.

[0091] RISC processors, such ARM processor, are strongly based on register operations, i.e., each data transfer is carried out through one of the internal registers of the CPU. To address such faults of processor **50**, a set of registers of the Fault Recognition Accelerator **11** is used as shadow copy of the processor **50** registers, and an extended visibility of these registers is obtained either using, if the extended core visibility needed to implement the first one is not available, a hardware FRAC module called Fault Recognition Accelerator-Core interface or a Fault Recognition Accelerator-Core software Interface, that will be detailed in describing modules **21** to **24**.

[0092] In particular, through such Fault Recognition Accelerator-Core interface, each time the Fault Recognition Accelerator **11** detects a write operation on one of the processor registers, it copies the written value in the corresponding shadow register. This is also done for Fault Forecasting: periodically, the CPU data will be compared with its shadow copy stored into the Fault Recognition Accelerator **11**.

[0093] Concerning faults in memories and data/instruction caches where no ECC/BISR is used, the Fault Recognition Accelerator Compiler **111** analyses the source code and compute for each variable a reliability weight, which will take into account the variable lifetime and its functional dependencies with other variables. The evaluation performed in this phase identifies a list of the most reliability-critical variables. Then, Fault Recognition Accelerator **11** will be configured to manage the most critical variables and it will follow and shadow them during their life in the microcontroller. Of course the user can also select which variables to protect.

[0094] An other important technique implemented in the proposed microcontroller is the program and flow signature on-line monitoring. For implementing robustness against code-faults the Fault Recognition Accelerator **11** generates run-time signatures of the code running on the CPU core **51** without overloading the processor and it checks the calculated signatures comparing them with the compile-time generated one. Through the Fault Recognition Accelerator Compiler **111**, the Fault Recognition Accelerator **11** can be configured to handle either control flow and/or data processing signatures. For Control flow signatures during the compile process, the target assembly code is split in branch-free blocks. A branch-free block is a set of consecutive instructions not containing jump. At this point, the Fault Recognition Accelerator Compiler **111** computes a signature composed by different sections:

[0095] a block signature, i.e., a signature of all the op-codes contained in each branch-free block. This signature is computed by the compiler by coding the operating codes of the instructions contained in the block by using self-consistent non-overlapping codes;

[0096] a control-flow regular expression. During compile time, a Control Flow Graph is computed and therefore information about allowed path respect previous block are included in each signature.

[0097] At execution time, the Fault Recognition Accelerator **11** will monitor the code execution flow. In particular, it will real-time compute and check the branch-free block signatures and it will real-time compare the executed program flow with the reference, either stored in the Fault Recognition Memory **12** or embedded in the user code by the compiler **111**. This approach will allow covering code faults, regardless their hardware location (memory or CPU). In case of error, the Fault Recognition Accelerator **11** will try (if possible) to repair the fault, restarting the execution flow from the last fault-free block (retry). Therefore, from the point of view of Fault Recognition Accelerator **11**, at the beginning of each block, the Fault Recognition Accelerator will recognize the starting signature and it will store this signature in an internal register for the run-time computing. At the end of the block the Fault Recognition Accelerator **11** will recognize either the branch instruction or the beginning of a new block (through its signature). In both cases, the run-time generated signature is compared with the expected result. When a branch occurs, the Fault Recognition Accelerator will recognize the next operation as one of the allowed signatures associated with the corresponding block. Therefore, in case of error, different actions can be taken: notify the error to the operating system, restart the program, jump to a previous check-point block.

[0098] The Fault Recognition Accelerator **11** in the proposed microcontroller also implements a data processing legality check procedure. Previous code and flow signature mechanism allow the coverage of code faults: it catches misinterpreted data, erroneous calculations, transient or permanent faults that cause any differences of the code execution flow with a very high coverage due to the use of the operation code as signature seed. However, this error detection mechanism can be further enhanced by adding protection on critical variables of the application running on the processor. With the same reliability-weight techniques used

by Fault Recognition Accelerator Compiler **111** to individuate memory variables to be shadowed, the most important variables of the application code (plus the one selected by the user) can be selected and coupled With either:

[**0099**] Legal Absolute values (i.e., maximum and minimum allowed values) that are checked by Fault Recognition Accelerator **11** during runtime execution;

[**0100**] Legal Delta values, evaluated with respect to the previous operation on the same variable. In such a way, the result of each single operation carried out by ALU or MAC can be run-time checked against the set of allowed values, and in the case of an error, the operation can be repeated or the execution stopped;

[**0101**] Legal Context values, i.e., timeframe and processes/tasks in which these variables can be read/written.

[**0102**] The same data processing legality check is performed by the hardware verification components **15** for variables living outside the CPU core and for signals exchanged with the external world.

[**0103**] An other important implemented procedure is the address legality check. During each data transfer, Fault Recognition Accelerator **11** checks legal values for a given address (Legal absolute, delta and context value) based on register to memory visible data transfers. When a mismatch is detected, an exception is generated. This is also performed in multi-process applications: based again on register to memory visible data transfers, Fault Recognition Accelerator **11** is able to performs checks on the memory accesses made by the various tasks, especially when virtual addressing is not used. More in detail, in case a task performs an access to non-shared or other tasks regions, an exception is generated.

[**0104**] An ALU concurrent integrity checking procedure is also proposed. The other crucial block in a RISC CPU is the integer core, in particular the Arithmetic Logic Unit (ALU), that often includes a Multiply-and-Accumulate (MAC) unit. Fault Recognition Accelerator **11** includes a reduced version of the ALU/MAC working with a coded copy of the shadowed registers (i.e., with a lesser number of bits), in order to concurrently check the results of the CPU core ALU/MAC without having a full copy of the ALU/MAC itself. In order to do that, extended visibility of the ALU/MAC core path is obtained either using the hardware FRAC module called Fault Recognition Accelerator-Core interface (detailed afterwards) or the Fault Recognition Accelerator-Core Software Interface (detailed afterwards) if the extended core visibility needed to implement the first one is not available.

[**0105**] In the proposed microcontroller, coding is implemented in such a way to maintain either basic mathematical operators such as sum, subtraction and multiplication, but also rotation, shift and similar logic operations as performed in a modern RISC processor.

[**0106**] A concurrent mode/interrupt check procedure is also implemented. During normal operation the Fault Recognition Accelerator **11** verifies flags and CPU mode signals, and on the basis of shadow Registers contents decides if the proper mode is set or not. When a mismatch is detected, an exception is generated. The same check is performed with interrupts, i.e., if interrupts were generated when not allowed or the opposite.

[**0107**] It is worth noting that the same circuit architecture based on the Fault Recognition Accelerator **11** and related Fault Recognition Compiler **111** can be also applied in same way to standard solutions, as for instance dual-redundant architecture as represented in **FIG. 3**. In such an architecture a central processor **51** operating in association with the Fault Recognition Accelerator **11** is coupled with a secondary processor **551**, operating in association with a second respective Fault Recognition Accelerator **511**. The two Fault Recognition Accelerators **11** and **511** are interacting using one of the Fault Recognition Accelerator modules **21** described afterwards. A comparator **501** is also included to select which of the processing units is currently selected based on decisions taken by the two Fault Recognition Accelerators **11** and **511**.

[**0108**] In the following, the modules that can be part of the Fault Recognition Accelerator **11** are detailed.

[**0109**] System modules **21** comprise:

[**0110**] a Global Supervisor Module that is the main controller and scheduler, for all the Fault Recognition Accelerator **11** activities, for instance the scheduling of BISTs sequence of **FIG. 3**. It also contains all the Fault Recognition Accelerator status and configuration information, to be shared with the Operative System and User Application code through the CPU system bus **53**;

[**0111**] a Program Watchdog Module that is composed by a watchdog triggered both by events inserted by the user into the software application and by time-windowed events automatically inserted by the Fault Recognition Accelerator Compiler **111** itself. For the events inserted by the Fault Recognition Accelerator Compiler **111**, the timeout event is computed taking into account expected timings limits (upper and lower). A timeout event generates an exception of this module for the Global Supervisor Module and to a Safe State Module, described in the following;

[**0112**] Fault Recognition Accelerator-Peripheral Manager Module, that is a module that manages all the interfaces of the peripherals of the Fault Recognition Accelerator **11**, e.g., the interfaces of hardware Verification Components **15**. When an error is detected in the CPU core **51**, peripherals can be placed in a safe state configured at compile-time, or when an error is detected in the peripherals, the Global Supervisor Module and the Safe State Module are informed;

[**0113**] FRAC to FRAC Module that is a module needed when additional system fault-detection is a requirement or if multi-Fault Recognition Accelerator architectures are performed. Such a module comprises substantially a block that interfaces the internal Fault Recognition Accelerator **11** with an external Fault Recognition Adcelerator (for hierarchical fault-robust systems or dual redundant systems as in **FIG. 3**) or with an external watchdog to

have an additional protection in case of Fault Recognition Accelerator faults;

[0114] Debug-If Manager Module that manages all the data transfer from Fault Recognition Accelerator **11** to the external world for debug purposes, through a JTAG or IS09141 interface.

[0115] Fault Recognition Accelerator-Memory Manager Module that manages all the interactions memory between the Fault Recognition Accelerator **11** and-the Fault recognition Memory **12**.

[0116] Clock Supervisor Module, that is a module comprising a clock guardian fed by an independent clock source. By way of example, this module can be implemented by counting the number of system clocks contained in a period of a slow clock oscillator, that can be internal or external to the Fault recognition accelerator **11** and the guardian is able to detect the degradation of the main clock during a fixed time window, i.e., if the clock is greater or lower of pre-determined safety levels. A timeout event generates an exception of this module for the Global Supervisor Module, and the slow clock is used to clock the Safe State Module to drive the system toward a fail-safe state;

[0117] Power Supervisor Module, that is a module composed by a scheduler that receives both from an external Power Supply Level Detector and from the Global Supervisor Module a signal specifying the need of a safe power-up or power-down. When enabled, it starts the Safe State Module and then provides a safe procedure to switch-on (power-up) or switch-off (power-down) the resets and clocks of the system. During the switch-off this module can also activate an external secondary power unit.

[0118] The Main Protection Modules **22** comprise:

[0119] Shadow Variable Module that makes a copy of the critical variables used by the application and updates such variables based on register to/from memory visible data transfers (Load/Store instructions). The shadow variable module compares memory to register transfers with the mismatches. When a mismatch is detected, an exception is generated for the Global Supervisor Module, the Safe State Module and the Data Retry Module, either hardware or software. The shadow variable modules can be configured via software to select variables to be protected.

[0120] Legal Variable Module, that is a module checking legal values for a given variable (Legal absolute, delta and context value) based on register to/from memory visible data transfers (Load/Store instructions) When a mismatch is detected, an exception is generated for the Global Supervisor Module and the Safe State Module. Legal variable module can be configured by software to select variables to be protected and values for the legal values;

[0121] Code&Flow Signature Module, that is a module, based on the code and flow signatures computed in compilation time by the Fault Recognition Accelerator Compiler **111**, and stored in the Fault Recognition Memory **12** or embedded in the user code memory, and based on code and flow signatures computed run-time by their module itself using the runtime instruction code. This module is able to detect faults in the CPU program flow. Furthermore, the module is able to detect faults in inline literals when they are loaded in the CPU core **51**. When a mismatch is detected, an exception is generated for the Global Supervisor Module, the Safe State Module and the Program Retry Module;

[0122] Safe State Module, that, when triggered by a software signal, or by the Global Supervisor Module, or by other modules as described above, is able to start a procedure to bring the system to a safe-state, i.e., by putting safe values in the variables, stopping the CPU **51**, sending a signal to the Fault Recognition Accelerator-Peripheral Manager Module to put all the peripherals of Fault Recognition Accelerator **11** in a safe state. It is configured at compile time;

[0123] Shadow Register Module, that makes a copy of the register bank of the CPU, and updates it based on memory to register visible data transfers (Load/Store instructions) and based on the access to the ALU Write pbrt available from Fault Recognition Accelerator-Core Interface Module. Then the shadow register module compares final register to memory transfers with the contents of the internal copy, detecting mismatches. When a mismatch is detected between internal results and the one on the ALU write port, an exception is generated for the Global Supervisor Module, the Safe State Module and the Data Retry. This module can be configured by software to select register to be protected;

[0124] Legal Load-Store Unit (LSU) Module, that is a module for checking legal values for a delta value, space and time context value) based on register to memory visible data transfers (Load/Store instructions). When a mismatch is detected, an exception is generated for the Global Supervisor Module and the Safe State Module. This module can be configured by software to select addresses to be protected and value for the legal values;

[0125] Task Supervisor Module, used in multi-process applications. Based on register to memory visible data transfers (Load/Store instructions), it is able to performs checks on the memory accesses made by the various tasks, especially when virtual addressing is not used. More in detail, in case a task performs an access to non-shared or others generated for the Global Supervisor Module, the Safe State Module and the Program Retry Module;

[0126] Data Retry Module, used when, besides fault-detection, also fault-correction is needed. Based on the results of the Shadow Variable or Shadow Register Module or Legal Variable Module, it is able to restore the value of a faulty variables/registers in the memories, by using the CPU ports, or by interacting with the Data Retry software module;

[0127] Prog Retry Module, used when besides fault-detection is needed also fault-correction. Based on the results of the Code&Flow Signature Module, it is

able to restore the CPU state and program counter, by using the CPU ports or by interacting with the Program Retry Module;

[0128] Fault Recognition Accelerator CPU Generic Modules **23** comprise:

[0129] Opcode Decode Module that is the generic module that decodes op-codes read from an interface between Fault Recognition Accelerator **11** and the Coprocessor **52** and from Fault Recognition Accelerator ETM (Embedded Trace Module) interface, and provides the decoded control signal to the other modules;

[0130] Exc/Int Supervisor Module, that is a module for continuously checking if the issued interrupts and exceptions are valid and served, considering the memory map configuration (access permissions, alignment and unaligned accesses etc.) and the interrupt requests. Furthermore it detects if an exception was supposed to be raised (for instance due to an user access to privileged areas) but it is not. If a fault is detected an exception is generated for the Global Supervisor Module and the Safe State Module;

[0131] Mode/Flag Supervisor Module, that verifies flags and CPU mode, based on Shadow Register Module and Fault Recognition Accelerator Software Register Shadowing Module, and based on signals coming from the CPU. When a mismatch is detected, an exception is generated for the Fault Recognition Accelerator Global Supervisor and the Fault Recognition Accelerator Safe State Module;

[0132] Boundary Supervisor Module, that continuously compares addresses, data and control signals generated by the Load-Store Unit (LSU) with the CPU memory boundaries, not checked by other modules, in order to detect faults in the units in between. When a mismatch is detected, an exception is generated for the Global Supervisor Module and the Safe State Module;

[0133] ALU/MAC Supervisor Module, that makes a reduced copy of the CPU Integer Core, including most critical ALU/MAC operations implemented on coded values (e.g., operating on module-three numbers). It has access to the Shadow Register Module and checks results detected on the ALU write port of the core, obtained through the Fault Recognition Accelerator-Core interface. When a mismatch is detected between internal results and the one on the ALU write port, an exception is generated for the Global Supervisor Module and the Safe State Module. If fault-detection is needed on operations not implemented in this reduced module, they can be decomposed in safe instructions by the Fault Recognition Accelerator Compiler **111**;

[0134] Coprocessor Supervisor Module, based on the Shadow Register Module, checks if the data transferred from the CPU registers to the coprocessor is valid or not. When a mismatch is detected, an exception is generated for the Global Supervisor Module and the Safe State Module;

[0135] BIST Supervisor Module that, when triggered by a software signal, or by the Global Supervisor Module, or by the Power Supervisor module, is able to start a BIST (Built In self Test) of the CPU core **51** or part of it. When a mismatch is detected, an exception is generated for the Global Supervisor Module and the Safe State Module;

[0136] Instruction Prediction Module, that is an advanced module carrying out a sort of a cache storing recent part of codes executed by the CPU core **51**, comparing with the current program flow and detecting differences. When a mismatch is detected, an exception is generated for the Global Supervisor Module, the Safe State Module, the Program Retry Module (Hardware or Software).

[0137] Fault Recognition Accelerator CPU Specific Module **24** comprise:

[0138] Fault Recognition Accelerator-INT module, that interfaces Fault Recognition Accelerator **11** with the CPU interrupt lines;

[0139] Fault Recognition Accelerator-COPROC interface module interfaces Fault Recognition Accelerator with the CPU Coprocessor **52** interface;

[0140] Fault Recognition Accelerator-Embedded Trace Module (ETM) Interface, that interfaces Fault Recognition Accelerator with the CPU core ETM Interface;

[0141] Fault Recognition Accelerator-Boundary Interface that interfaces Fault Recognition Accelerator **11** with the CPU buses **53** and **63** and interface signals, such as Instruction Cache Memories, AMBA busses, and so on.

[0142] Fault Recognition Accelerator-Core interface, that interfaces Fault Recognition Accelerator with the CPU Integer Core to give extended hardware visibility (such as the visibility of the ALU Write port for the register shadowing, and so on).

[0143] For what concerns the Fault recognition Acceleration Compiler **111**, the Initialization software Modules **112** contain the code needed to initialize the Fault Recognition Accelerator **11** with respect to the system.

[0144] The Operative System Interaction software Modules **113**, that are needed to interact with the Operative System, comprise an Operative System Driver Software Module containing the code needed to configure Fault Recognition Accelerator **11** with some Operating System runtime settings, such as the addresses where an image is mapped at runtime, dynamic Process ID. Furthermore, the module comprises some code possibly needed by the operating system to read Fault Recognition Accelerator fault status, as well as other statistic values. This interaction is to release Fault Recognition Accelerator status information to the Operative System.

[0145] Application Configuration Software Modules **114**, needed to configure the Fault Recognition Accelerator **11** comprise:

[0146] Base Configuration Software Module containing the code needed to configure the Fault Recognition Accelerator **11** with respect to the Operative System and the Application User Code;

[0147] Signature Configuration Software Module, containing the code needed to configure the Fault Recognition Accelerator with the code&flow signatures computed at compile time by Fault Recognition Accelerator Compiler **111**.

[0148] Finally Runtime Modules **115**, as mentioned previously, are in most part the counter-part of already described hardware modules of the Fault Recognition Accelerator **11** hardware modules, i.e., they are needed only if the correspondent function is not realized in the correspondent hardware module. Runtime modules **115** comprise:

[0149] EXC/INT Supervisor Software Module, that is the software counter-part for the hardware Exc/Int Supervisor Module;

[0150] CPU Supervisor Software Module, that contains the code needed to verify the processor configuration (block settings, memory mapping etc.). If a fault is detected, the module is able to restore the program flow as the Program Retry Module;

[0151] Task Supervisor Software Module, that is the software counter-part for the Task Supervisor Module;

[0152] Fault Recognition Accelerator-Core Software Interface is composed by code that mimics at software level what Fault Recognition Accelerator-Core Interface does at hardware level. In particular, when necessary, copies of the internal core registers are made into the Fault Recognition Accelerator **11**;

[0153] Coprocessor Supervisor Software, that is the software counter-part for the hardware Coprocessor Supervisor Module;

[0154] BIST Supervisor Software Module, that is the software counter part for the hardware BIST Supervisor Module;

[0155] Data Retry Software Module, that is the software counterpart for the hardware Data Retry Module;

[0156] Program Retry Software Module that is the software counterpart for hardware Program Retry Module.

[0157] For what concerns the peripherals of the Fault Recognition Accelerator **11**, the Fault-Tolerant Memory interfaces **13** comprise:

[0158] a Fault Recognitor F-RAM Module, that wraps an external memory and code its contents in order to carry out an on-line Built-In Self Test and if needed on-line Built-In Self Repair;

[0159] a Fault Recognitor F-ROM Module that wraps an external memory and code its contents in order to carry out an on-line Built-In Self Test and if needed on-line 'Built-In Self Repair';

[0160] Fault Recognition F-DMA Module, that comprises a block that wraps the external memory DMA controller in order to check for address and data faults.

[0161] The Bus Interface Controller **14** comprises:

[0162] AHB Module that uses a redundant coded AHB bus in order to check data transmitted on the main AHB (Advanced High-performance Bus) bus like the bus system **53**;

[0163] APB Module, that uses a redundant coded APB bus in order to check data transmitted on the main APB (Advanced Peripheral Bus) bus, like peripheral bus **63**;

[0164] CAN Module that implements extra levels of fault-detection and correction for CAN bus modules.

[0165] As described above hardware verification component **15**, are used for applying distributed fault tolerance techniques to the remaining interfaces and sensors of the microcontroller **10**.

[0166] Hardware Verification Components **15** can be thought as the hardware realization of basic monitoring functions of the SpecMan Elite e-Verification Components (eVC), used in the design method that will be detailed with reference to **FIGS. 6 and 7**.

[0167] A hardware Verification Component **15** can be viewed as the inverse function of the module function to be monitored. **FIG. 4** shows an example of the hardware Verification Component **15**, associated to a generic peripheral PHY, that is interfaced both to the system bus **53** and directly connected to primary outputs.

[0168] The peripheral PHY generates three output signals: a clock C, and two outputs A and B. The main constraint on peripheral PHY is that output B must not change after three cycles from the change of A. The "e" language construct to check this condition is:

```
Expect@A_rise=>{[1.3];@B:rise} else PHY error
```

[0169] In this way, independently from the correctness of the software or hardware that are currently driving the output ports of the peripheral PHY, an error can be generated in case of illegal situations. Errors are then collected and managed by Fault Recognition Accelerator **11**.

[0170] These kind of "e" language constructs can be easily implemented in hardware in a hardware Verification Component **15** with a watchdog and a comparator. The other side of peripheral PHY is instead verified by the system verification procedure.

[0171] It is clear that the hardware Verification Component **15** complexity can easily grow up depending on the complexity of the signals to be verified. However it is also clear that the procedure exploiting hardware Verification Component **15** can increase system reliability by allowing runtime verification of illegal output configuration. Moreover, the hardware Verification Component **15** can be automatically generated starting from the eVC components used for system verification, by using a library of watchdogs, state machines, samplers, and so on. The integration of eVC and hardware Verification Component **15** into the same verification procedure can be a powerful tool for standardized design in a platform environment.

[0172] Hardware verification components **15** are supplied with a suitable hVC design kit, i.e., a software tool that

translates the e-language verification constraints in Hardware verification components, by using:

[0173] a set of hVC base modules, that are detailed with reference to **FIG. 5**;

[0174] an hVC library of precompiled hardware verification components for standard blocks such PWM, timers and so on.

[0175] As shown in **FIG. 5**, the Fault Recognition Accelerator **11** communicates through the Fault Recognition Peripherals Manager Module belonging to modules **21** with the base hardware Verification Component **15** for monitoring an actuator **606** and hardware verification component **15'** for monitoring a sensor **607**.

[0176] The hardware verification component **15** comprises:

[0177] a Hardware Verification Checker Module **604**, that embodies the hardware-verification checking unit of the Hardware verification block **15** for a given interface;

[0178] a Hardware Verification BIST Module **605**, that embodies the hardware-verification BIST unit of the hardware Verification Component **15** for a given interface;

[0179] a Hardware Verification Interface Module **601**, performing a safe connection between the Fault Recognition Accelerator **11** and the hardware Verification Components **15**, receiving configurations values, providing Fault Recognition Accelerator **11** with go/no-go signals, receiving commands from Fault Recognition Accelerator **11**, interacting with the Fault Recognition Peripherals Manager Module;

[0180] a Hardware Verification-Manager Module **602**, that is the manager of the whole hardware Verification Component **15**, and it is the responsible for putting the controlled interface in a safe state when needed;

[0181] Hardware Verification-MUX Module **603** that is a safe multiplexer structure used to select between system signal, for instance, central processing unit **51**, and hardware Verification Components **15** signals;

[0182] a Hardware Verification-Configuration software Module that implements at compile time or design time all the needed features to configure the modules of hardware Verification Components **15**, e.g., selecting/ordering legal-delta-context checking for the interfaces, etc. This module is part of the Initialization Software Modules previously described for the Fault Recognition Compiler **111**.

[0183] As it can be seen from **FIG. 5**, hardware Verification Component **15'** for the sensor **607** does not need the Hardware Verification BIST Module **605**.

[0184] As mentioned above, the microcontroller **10** comprising the fault recognition accelerator **11** and its peripherals is designed by using a design procedure able to apply fault-avoidance and fault-removal techniques in order to achieve the desired system dependability. Such a design procedure comprises not only functional verification, but also fault-modeling, fault-injection and fault-forecasting at the design level that are integrated in the same design verification flow.

[0185] The proposed design procedure comprises a fault-injection method for testing the fault tolerance of the system with respect to a specified behavior, i.e., injecting faults in the design to validate and assess the fault-robustness of the designed system.

[0186] In association with the proposed fault-injection method a procedure called e-Fault, based on Verisity Specman Elite tool, is here described. Such a procedure is able to test for all the faults both at the device level (i.e., RTL or gate-level) and at the system level.

[0187] Summarizing, the design and verification procedure of an Embedded Computational Unit (ECU) based on the proposed microcontroller is mainly composed by:

[0188] a set of eVC components for the platform components. Using these building blocks more complex eVC can be composed in order to verify system-level aspects;

[0189] a CPU verification procedure. It is mainly composed by a constraint-driven test generator, that generates corner-case instruction sequences based both on user constraints and feedback from the coverage analyzer;

[0190] a fault injector implemented using the capabilities of SpecMan Elite;

[0191] a database of typical system situations, in particular automotive system situations, such as: multi-processor CAN/TTP units, fault models, and so on.

[0192] Such a design procedure can be easily integrated with a design-flow for automotive systems, resulting in a flow as depicted in **FIG. 6**, where:

[0193] step **301** represents taking in account specifications;

[0194] step **302** represents system definition;

[0195] step **303** represents system partitioning and simulation;

[0196] step **304** represents fault-tolerance system definition;

[0197] step **305** represents ECU design;

[0198] step **306** represents ECU verification;

[0199] step **307** represents system verification with fault injection;

[0200] step **308** represents final implementation.

[0201] System definition **302**, system partitioning and simulation **303** at highest level are performed in SimuLink/Matlab environment, whereas ECU design **305**, ECU verification **306** and system simulation/fault-injection **307** steps are performed in the proposed design and verification procedure. In particular the following steps are related to this invention:

[0202] Fault Tolerance System Definition **304**: this step can be performed both at RTOS (Real-Time

Operating System) and system level starting in the SimuLink environment by using standard Hazard Analysis techniques, such as Software Failure Modes and Effects Analysis or others. Then, when basic needs are defined, such as for instance definition of fail-silent or fail-safe behaviors, these choices can be implemented at ECU design level within the design procedure, by defining for instance the level of Fault Recognition Accelerator **11** hardware/software integration for each of the system components, the dimension of the Fault Recognition Accelerator memory **12**, and so on.

[**0203**] ECU design **305** and verification **306**: after system partitioning and fault-tolerance techniques definition, each block can be designed and verified, together with eVC and fault injection;

[**0204**] System Verification with Fault-injection **307**: the complete system can be verified against faults and the fault-tolerance choices can be validated at the system level.

[**0205**] It is worth noting that both the modularity of the design procedure and the possibility to verify fault-tolerant strategies together with the system specification, strongly reduce design time by dramatically reducing fault-avoidance and fault-removal time.

[**0206**] The procedure for implementing the fault-injection steps **306** and **307**, will be now described. Such a procedure is based on the SpecMan Elite verification tool from Verisity Corporation, a verification technology that provides the above mentioned re-usable e-Verification Components eVC, written in a proprietary high level 'e' language. eVCs incorporate the test stimulus and expected results within them. An advantage of this verification technique is the portability and re-usability of eVCs.

[**0207**] **FIG. 7** schematically represents said verification tool, where a System under Test **410** is tested through a general test bench **411**, that receives as input data from an eVC database **412**, operating according the usual SpecMan Elite verification tools, and from an e-Fault injector **400**. Such an e-Fault injector **400** comprises a Faults Database **406**. Such a Faults Database **406** is extended either with additional Faults Packages **405** or customized by the user by means of a Fault-Modeler **40**. The e-Fault injector **400** further includes a Fault-List Generator **407** able to select and compact the faults to be injected in the System under Test **410**, a Fault-Injector Engine **402** with advanced fault injection algorithms, a Test Generator **403** to provide to the test bench **411** standard test patterns for fault sensitivity measurement, and a detailed Fault-Analyzer **401** in order to have statistics on fault effects. An IEC 61508 Checker **408** is implemented on a output **413** to verify compliance with IEC 61508 normative for safety critical design of electronic systems.

[**0208**] Faults Database **406** includes the following base fault models:

[**0209**] basic "bit-level" models such stuck-at, open line, indetermination, short, bridging, glitches and bit-flips;

[**0210**] basic memory fault models (SRAM, ROM, and so on);

[**0211**] faults in the clock/power lines;

[**0212**] extension of "bit-level" fault models at higher-level of abstraction (e.g., pure behavior), such variable faults, branches/cases conditions mutants, and so on.

[**0213**] In addition, special Faults Packages **405** are realized-with advanced radiation effect fault models (SEU package), typical fault models for bus protocols (BUS package, with CAN, OCP, AMBA faults), faults for standard digital (CPU package, with APB/AHB slaves), advanced memories configurations (MEM package, with Flash, D-RAM), ARM package with fault models for ARM processors (i.e., with models to insert faults in instructions through the modification of the correspondent micro-code), and so on.

[**0214**] The Fault Modeler **404** is also provided to add custom fault models. Models are fully configurable and main parameters are physical parameters for faults (shape, duration, amplitude, cardinality, etc.), frequency of faults, faults probability distribution.

[**0215**] Fault-List generator **407** is a pre-processor that is included to generate the fault list and collapse it based on static and dynamic collapsing algorithms, to reduce simulation time. User is also able to control parts of the system to be injected and add extra constraints for the fault list generation.

[**0216**] Test Generator **403** is able to generate the workload for device-oriented fault-injection or more in general to increase the fault coverage of a mission-oriented fault-injection covering the part of the device under test not covered by the mission workload.

[**0217**] Fault-injector Engine **402** is an engine written in e-language to realize the algorithm to inject faults in the system. User is able to control fault injection for instance by choosing random, pseudo-random, deterministic fault injection.

[**0218**] Results of fault simulation are analyzed by the Fault Analyzer **401** with basic comparison rules (such as device state at sample points, IO values, etc.) and user defined comparison rules. Fault Analyzer **401** includes a Collector Module to store information about the simulations that have been run. This information is organized in a graphical way, e.g., about how many of the injected faults have been well tolerated, how many have produced errors, and each graphical representation is plotted for fault type, system block, and so on. Results are measured in sensitivity, FIT rates, propagation/detection/recovery latencies.

[**0219**] Finally, an IEC 61508 Checker **408** is included to allow the injection of standard faults defined by the IEC 61508 norm (such as faults for Bus, CPU, clocks, and so on) and the analysis of fault effects based on IEC 61508 safety integrity levels requirements).

[**0220**] The arrangements disclosed herein permit to obtain remarkable advantages over the known solutions.

[**0221**] The use of a mixed hardware and software approach to design fault-robust systems, implementing hardware redundancy controlled and configured by software, allows to achieve fault tolerance of complex CPU-based system on a chips without compromising execution time and code dimensions and with an acceptable hardware overhead.

[0222] The microcontroller according to the invention advantageously is based on a full-configurable asymmetric redundancy, reducing remarkably the area overhead compared with dual or multiple redundancies. Thanks to its limited area overhead, such a microcontroller can be built with gate-level fault-tolerance techniques (such triple-modular-redundancy flip-flops in state machines) in order to have the highest degree of robustness and negligible probability of unrecoverable internal failures.

[0223] It is intrinsically a distributed fault-tolerant approach, where fault-tolerant techniques and associated circuits are distributed in every part of the microcontroller, watching a particular problem and communicating with a central fault-tolerant processor that coordinates all the peripherals and keeps track of the fault-history of the system. Thanks to the strict interaction between the central fault-tolerant processor and the CPU, the system is also a low-latency fault-detection system, i.e., faults are caught very soon to avoid error propagation and to allow earlier safe system recovery. Finally, the system is customized and customizable: thanks to its strict hardware-software interaction, it gives many possibilities to the application and Operative System designers in order to configure the part of the application code or set of variables or peripherals to be protected, including as much as fault tolerance they need.

[0224] Without prejudice to the underlying principle of the invention, the details and embodiment may vary, also significantly, with respect to what has been discussed just by way of example without departing from the scope of the invention, ad defined by the claims that follow.

[0225] The primary field of application of the arrangement disclosed herein are microcontrollers or System on Chip for automotive applications, but it is clear that the scope of the invention extends to achievement of dependability in all systems comprising a microcontroller, comprising a central processing unit, a system bus and one or more functional parts comprising interfaces and peripherals, where a further fault processing unit suitable for performing validation of operations of said central processing unit is provided.

[0226] For instance, for what concerns safety applications, the present invention is applicable not only in safety-critical systems (such as automotive, aerospace and biomedics) but also in reliability or availability critical systems such high-speed communications or advanced multimedia systems.

[0227] In terms of faults respect to which the system should be robust, the present invention is not limited to a particular kind of faults. The invention is addressing either:

[0228] mechanical faults, i.e., due to deterioration (wear, fatigue, corrosion) or shock (fractures, striction, overload), as soon as they cause an electronic fault in the system;

[0229] Electronic faults, i.e., due to latent manufacturing defects (stuck-at in hidden part of the circuit), or to the operating environment (radiation, noise, heat, ESD, electro-migration), as soon as they are detected by the fault-detection mechanisms and, concerning design defects, as soon as they cause either an illegal execution-flow and/or an illegal operating code processing and/or an illegal values of critical variables, respect to the application code;

[0230] Software faults (including the ones in the Compiler), i.e., design defects or "code rot" (e.g., accumulated run-time faults), if they cause either an illegal execution-flow and/or an illegal operating code processing and/or an illegal values of critical variables, respect to the application code. Software faults are also detected if they cause illegal transitions in peripherals.

[0231] Moreover, the present invention can detect permanent faults but also both transient and intermittent ones. Concerning transient faults, it is worth noting that in modern DSM technologies transient faults (often called Soft Errors) can be very frequent. Soft Error Rate (SER) has traditionally concerned the nuclear and space communities but it is increasingly worrying semiconductor companies and designers of consumer and communication products.

[0232] All of the above U.S. patents, U.S. patent application publications, U.S. patent applications, foreign patents, foreign patent applications and non-patent publications referred to in this specification and/or listed in the Application Data Sheet, are incorporated herein by reference, in their entirety.

[0233] From the foregoing it will be appreciated that, although specific embodiments of the invention have been described herein for purposes of illustration, various modifications may be made without deviating from the spirit and scope of the invention. Accordingly, the invention is not limited except as by the appended claims.

1. Dependable microcontroller comprising:

a central processing unit,

a system bus and one or more functional parts interfaces and peripherals; and

a further fault processing unit suitable for performing validation of operations of said central processing unit,

said further fault processing unit being external and different with respect to said central processing unit, and said further fault processing unit comprises at least a module, operating at least partly on-line, configured for performing validation of operations of said central processing unit and one or more modules operating at least partly on-line, configured for performing validation of operations of said other functional parts of said microcontroller.

2. The microcontroller of claim 1 characterized in that said one or more modules suitable for performing validation of operations said other functional parts comprise system modules suitable for interacting with fault recognition peripherals associated to said further fault processing unit, for interacting with basic system components and for managing other internal modules of the further fault processing unit.

3. The microcontroller of claim 2 characterized in that said system modules include a global supervisor module for managing other internal modules of said further fault processing unit.

4. The microcontroller of claim 3 characterized in that said system modules include a FRAC to FRAC interface unit suitable for exchanging informations with a second fault processing unit or with an external watchdog.

5. The microcontroller of claim 1 characterized in that said further fault processing unit is performing validation of operations of said central processing unit by using one or more of the following fault protection techniques:

data shadowing;

code and flow signature;

data processing legality check;

addressing legality check;

ALU concurrent integrity checking;

concurrent mode/interrupt check.

6. The microcontroller of claim 1 characterized in that said at least a module for performing validation of operations of said central processing unit comprises main protection modules, suitable for implementing said fault protection techniques.

7. The microcontroller of claim 6 characterized in that main protection modules comprise one or more among the following modules:

Shadow Variable Module suitable for making a copy of the critical variables used by the application;

Legal Variable Module, suitable for checking legal values for a given variable;

Code&Flow Signature Module, suitable for detecting faults in the CPU program flow on the basis of code signatures;

Safe State Module, suitable for bringing the system to a safe-state;

Shadow Register Module suitable for making a copy of the CPU register set;

Legal Load/Store Unit Module suitable for checking legal values for a given address;

Task Supervisor Module suitable for multi-process applications, checking legal memory accesses made by the various tasks;

Data Retry Module, suitable for implementing fault-correction by restoring values in faulty variables/registers in the memories;

Prog Retry Module, suitable for implementing fault-correction by restoring the CPU state and program counter.

8. The microcontroller of claim 1 characterized in that

said at least a module for performing validation of operations of said central processing unit comprises CPU interface modules suitable for exchanging data with the central processing unit and the system bus.

9. The microcontroller of claim 8 characterized in that said CPU interface modules comprise interfaces between said further fault processing unit and a central processing unit integer core, said interfaces carrying out a dedicated bus connected to said integer core, in order to ensure extended hardware visibility.

10. The microcontroller of claim 8 characterized in that said at least a module for performing validation of operations of said central processing unit comprises CPU modules suitable for processing data arriving from the Fault Recognition Accelerator/CPU interface modules and to exchange data with the main protection modules and in that said CPU modules comprise one or more among the following modules:

EXC/INT Supervisor Module, suitable for concurrently checking if the issued interrupts and exceptions are valid and served, considering the memory map configuration (access permissions, alignment and unaligned accesses etc.) and the interrupt requests;

Mode/Flag Supervisor Module, suitable for concurrently checking if flags and CPU mode signals are valid;

ALU/MAC Supervisor Module, suitable for concurrently checking the operation of the CPU core ALU/MAC;

BIST Supervisor Module, suitable to start a BIST of the CPU core or part of it;

Instruction Prediction Module, suitable to realize a sort of a cache storing recent part of codes executed by the CPU, comparing with the current program flow and detecting differences.

11. The microcontroller of claim 10 characterized in that said central processing unit and said further fault processing unit suitable for performing validation of operations are coupled with a secondary processing unit and a respective second fault processing unit suitable for performing validation of operations of the secondary processing unit in order to realize a dual redundant system.

12. The microcontroller of claim 1 characterized in that said fault recognition peripherals comprises a memory module.

13. The microcontroller of claim 1 characterized in that said fault recognition peripherals comprise fault tolerant memory interfaces.

14. The microcontroller of claim 1 characterized in that hardware verification components are provided associated to said other functional parts of said microcontroller for performing validation of operations under control of said one or more modules.

15. The microcontroller of claim 14 characterized in that said hardware verification components are obtained as hardware translation of codes (eVc) used for design verification.

16. The microcontroller of claim 15 characterized in that said hardware verification components comprise one or more among the following basic hardware verification modules that implement said hardware translation:

a module for hardware-verification checking;

a module for hardware-verification BIST;

a module for hardware verification of interfaces;

a module for hardware verification management;

a multiplexer structure for selecting between system signal and hardware Verification Components signals;

a module for configuration of software.

17. The microcontroller of claim 16 characterized in that said basic hardware verification modules are configured through an Hardware Verification-Configuration software Module, comprised in initialization modules of a compiling procedure.

18. The microcontroller of claim 1 characterized in that said fault recognition peripherals further comprise a bus interface controller for monitoring the system bus of said microcontroller.

19. The microcontroller of claim 1 characterized in that said fault recognition peripherals further comprise a simplified processor associated to application specific peripherals.

20. The microcontroller of claim 1 characterized in that implements a compiling procedure for performing at least one of the following operations:

initialization of basic functions of the further fault processing unit;

interaction with an operative system;

application configuration of the further fault processing unit, with respect to the operative system and a user application code;

insertion of run time modules in the user application code to allow fault-detection and protection techniques.

21. The microcontroller of claim 9 characterized in that said insertion of run time modules in application code implements direct CPU registers and ALU visibility, and/or tasks and exceptions supervising, and/or BIST tests.

22. Method for detecting faults in the microcontroller of claim 1 characterized in that said processing unit executes during said microcontroller activity on-line fault tolerance operations suitable for generating fault alarms and fault events if a fault is detected.

23. The method of claim 22 characterized in that said on-line fault tolerance operations implement one or more of the following fault protection techniques:

data shadowing;

code&flow signature;

data processing legality check;

addressing legality check;

ALU concurrent integrity checking;

concurrent mode/interrupt check.

24. The method of claim 22 characterized in that said fault processing unit executes during said microcontroller activity also the following operations in different times:

microcontroller fault-forecasting at power-on operations;

microcontroller on-line fault-forecasting operations.

25. The method of claim 24 characterized in that said microcontroller fault-forecasting at power-on operations comprise running a sequence of BIST tests or BISTS run at power-on.

26. The method of claim 24 characterized in that said microcontroller on-line fault-forecasting operations comprise executing a subset of said sequence of BIST tests or BISTS run during the system runtime.

27. Method for designing a fault tolerant system for a microcontroller, characterized in that includes the operations of:

performing a functional verification using software verification components (eVC);

performing a central processing unit verification procedure;

performing a fault injection procedure;

performing final implementation.

28. The method of claim 27 characterized in that further includes the operation of providing a database of typical system situations.

29. The method of claim 27 characterized in that said software verification components for the microcontroller interfaces are automatically translated in hardware verification components by an hVC design kit during said final implementation step.

30. The method of claim 27 characterized in that said fault injection procedure comprises a e-Fault phase operating in association with one or more among the following modules:

a Faults Database module, including a combination of base fault models;

a Faults Packages module, including a combination of advanced radiation effect fault models;

a Fault-Modeler module suitable for adding custom fault models;

a Fault-List Generator module for selecting and compacting the faults to be injected;

a Fault-Injector Engine module for providing injection procedures;

Test Generator for providing test patterns;

a Fault-Analyzer module to provide statistics on fault effects;

a Checker module for verifying compliance with normatives for safety critical design of electronic systems.

31. A computer program product directly loadable in the memory of a computer and comprising software code portions for performing the method of claim 27 when the product is run on a computer.

* * * * *